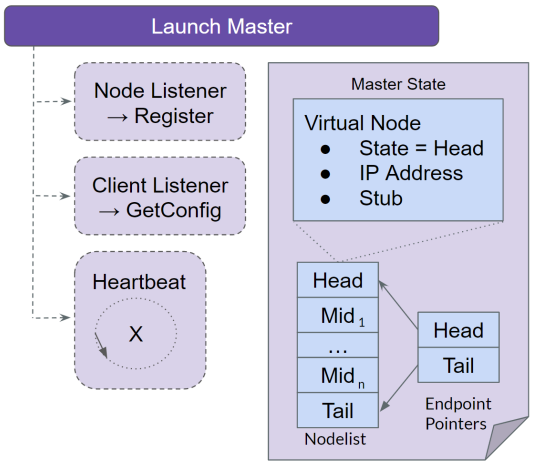
Introduction

We have implemented a replicated block store using chain replication. The replicated block store allows client applications to read and write 4K blocks just as they would with any other block device, except replicated at N remote nodes. As we are imitating the behavior of a block device, we wanted to provide linearizable semantics. By using chain replication, we were able to create a system that allows for an arbitrary number of nodes, without sacrificing throughput or consistency.

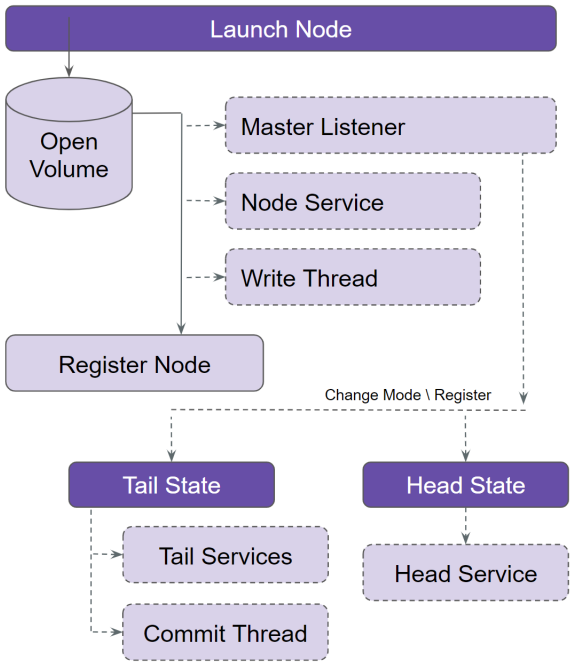
Architecture

Our implementation for a highly available chained replication system accommodates a flexible number of nodes in a chain ranging from 1 to N. The chain is managed by a master, which informs clients about chain endpoints (head/tail). Clients write to the head node and read from the tail. Our system is linearizable up to N-1 failures and durable and consistent even with total chain failure so long as the last node to fail is brought up first. We built our prototype in C and C++ using gRPC for communication.



The master coordinates the chain. It tracks nodes in the chain, informs nodes of state change and handles membership through registration and failure management. It also communicates endpoints to the client. The master stores an internal representation of the chain, along with communication and state details for each node.

Our nodes run a variety of threads and services to accomplish tasks and facilitate communication with other nodes, the master and clients. Each node stores communication details about the up and downstream nodes in the chain, along with its own state. Nodes launch as tail, and as state changes so do the services that are exposed.



Client Library

Client library is structured to be a C++ class that an user can initiate an instance and call functions to. The class takes in the master's ip address and the client's ip address, and uses the system library to get the process's identifier upon initialization. The client's ip and pid is used to construct an identifier that uniquely identifies the client. That unique identifier is

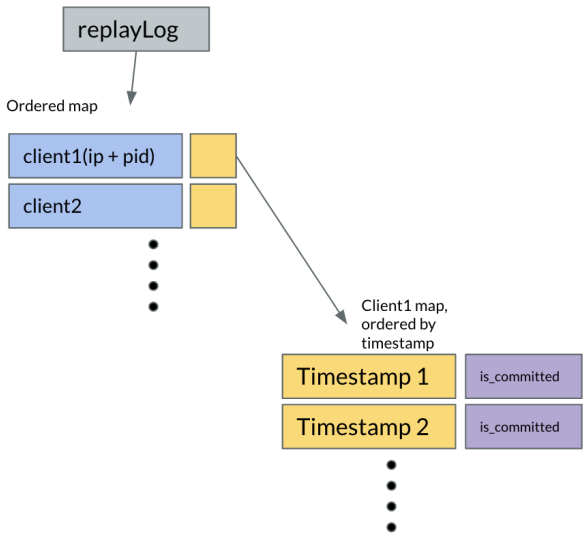
going to be used as a part of the *Client Request ID,* which uniquely identifies a request in the server space.

The client library is constructed in a way that allows sending asynchronous writes to the server quickly, and acknowledges whether those writes commit successfully later. The client can also choose not to acknowledge any write, if the user application prefers to verify whether the write went through in another way.

A private variable is used to track a list pending write. The write() call will add a pending write to the list. And if the user chooses to call AckWrite() to ack the write, the pending write entry will be removed if the return of the ack call shows the request has been committed on the server's end. There is also a function available for the user to remove the request with the oldest timestamp from the pending list, if they wish not to ack some writes at all. A RetryTopPendingWrite() function is also provided for the user to retry a write request with the oldest timestamp in the pending list.

Replay Log

The replay log is a structure each server keeps in memory. It is a two level map as shown in the diagram. The reply log is used to allow the server to detect duplicate requests(requests with the same *Client Request ID*). All the implementations of the replay log are abstracted away. Only defined interfaces are exposed for other logic to interact with the replayLog. It includes a function to add to the replaylog, which also returns error code if the entry to be added is already present in the log.



The reply log also tracked commit info, so a CommitLogEntry() is provided for the server node to call when a request has been committed in storage. When receiving a write ack call client, the reply log is used by the tail node to check whether the client request has been committed or not. A function AckLogEntry() is provided for that purpose.

There is also a thread safe function CleanOldLogEntry() provided. The function is intended to be used by a standalone thread to run periodically to clean out entries that are too old.

Experimental Setup

All benchmarks (including storage microbenchmarks) were run on a 6-node CloudLab cluster, with nodes configured with Ubuntu 20.04, 4 GB memory, 10-core 2.6GHz CPU, and

HDD storage.

We tested the consistency of our system under node failures using a similar programmatic crash approach as our P3 implementation. We created ‘crash code’ values for write offsets, encoded with a sequence indicating that the offset represented a crash code, a target server IP, and a numeric crash point. At various crash points in our server code, we could then insert checks on that offset value; if the server IP and crash point value both matched the server would crash, and if not it would proceed as if it were offset 0. This allowed us to reliably crash a single server node at a specific line in our code without impacting the operation of servers surrounding it on the chain, using only different client requests and no per-crash-scenario modifications to the server.

To assist with testing consistency, we also implemented a checksum function on our servers which, when called by a client on the tail, would compare the checksum of the committed volume for every node on the chain and indicate whether server data was consistent with each other. The checksums are calculated by the low-level volume implementation to hash the actual data. This way even if two nodes have commited writes in a different order or even use different implementations, they will give the same checksum. By having clients write data while deliberately triggering crashes and re-integrating nodes, then reading expected data and checking the checksums at each stage, we can establish that our system is able to handle failover and reintegration without inconsistency.

Storage

On each node, the volume is stored as a single file containing the sequence number of the most recently committed write and a list of 4K data blocks. Because the order of the data blocks in the file are unlikely to reflect their offset within the volume, each volume also keeps a metadata table mapping volume offsets to file offset. Each entry in the table also contains the sequence number of the most recent write to modify a block, which is needed for reintegration. The volume is implemented as a modular interface to make it easier to try different implementations.

We made two implementations, one with atomic commits and one with non-atomic commits. Both wait until all data is persistent on disk before updating the last commit number, so they are sufficient to ensure linearizability given node failures. Both implementations write new data to new blocks of the file instead of overwriting the existing blocks. For aligned writes, this simply means writing a single 4K block. For unaligned writes, it must first read and copy the unmodified part of each of the two blocks, and write two new blocks.

The non-atomic commit implementation uses a linear metadata table. While this takes more space on disk for sparse volumes, it allows for faster metadata lookup. On commit, the metadata table is updated to point to the new data block(s). This is fine for aligned writes, however unaligned writes, it is possible to end up with a partial write if the node crashes after updating one metadata entry, but before updating the other. This is sufficient to guarantee linearizability assuming at least one node stays alive, but we implemented the atomic commit version so that we could guarantee linearizability even if all nodes fail at once.

The atomic commit implementation uses copy-on-write for the metadata, and thus only has a single write to update metadata on commit. To minimize the amount copied per write for metadata, we used a multi-level table.

We measured the performance of the two volume implementations using microbenchmarks which are summarized below. Note that sequential vs random workloads did not show any difference. This is expected because the way blocks are allocated within the file result in all writes being random. We expected writes to be faster in the non-atomic implementation because the atomic implementation has to update metadata on write. Conversely we expected commits to be faster on the atomic implementation because the non-atomic commit has to write more metadata. We also expected reads to be faster on the non-atomic implementation because reading metadata is faster. Finally, we expected commits of old writes to be faster than write-commit together because there is more time for the kernel to push writes to disk before the fsync in commit.

	Atomic	Non-atomic
Aligned write	19 us	4.6 us
Aligned read	3 us	2.3 us
Unaligned write	25.6 us	10.5 us
Unaligned read	5.9 us	4.1 us
Aligned commit old	1.7 ms	2.5 ms
Unaligned commit old	1.7 ms	5.8 ms
Write-commit	12 ms	12 ms

Message Propagation

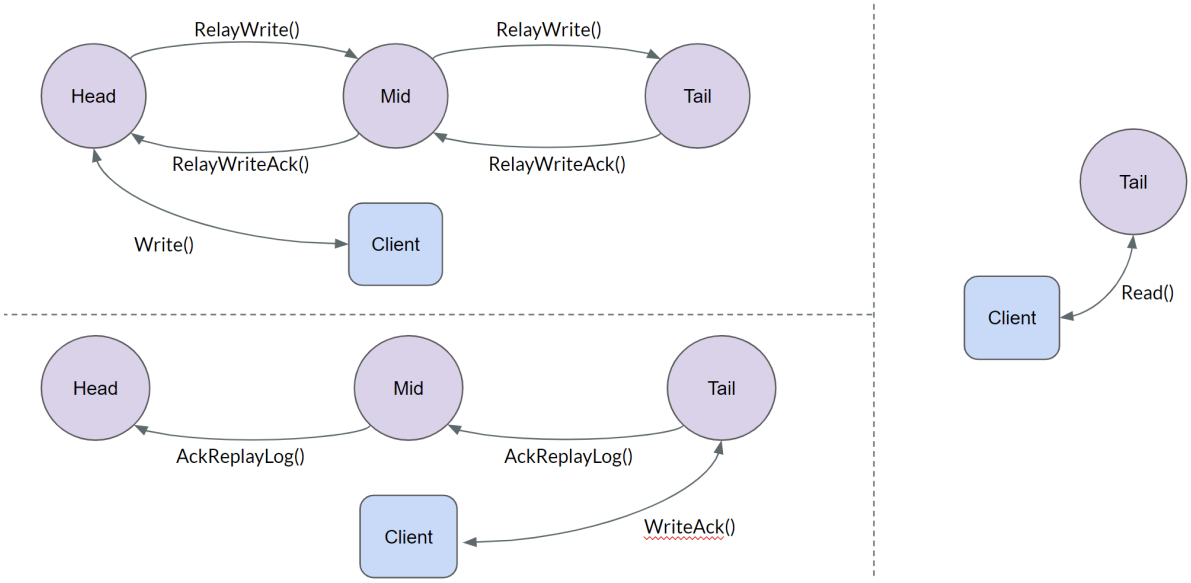
In our system, we follow the write propagation protocol outlined by the Chain Replication paper at a high level. Client requests are received by special services running on the head and tail nodes and then propagated downstream and upstream across the chain by a node listener service present on every node.

The head node service exposes a Write() function to clients, which adds requests to the node's pending list and replay log. This Write() function also checks incoming client request IDs against the replay log for eliminating retried duplicates.

All nodes contain a 'write' background thread which pulls items off their pending list, writes the data to volume (but doesn't commit), forwards the entry to a downstream node if

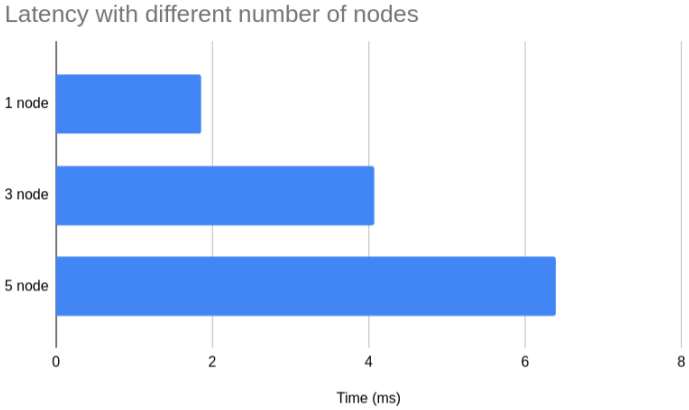
possible, and adds the entry to their sent list. All nodes also provide RelayWrite() grpc calls which act similar to Write() on the head but for inter-node communication, adding forwarded requests to the node's pending list and replay log; nodes also provide a RelayWriteAck() and AckReplayLog() function for upstream propagation of commits and replay log acknowledgements, respectively.

The tail node contains a 'commit' background thread which pulls items off the sent list, commits them to the volume, marks them as committed in the replay log, and propagates the commit upstream if possible. It exposes a Read() function to clients which simply returns read data, and a WriteAck() function that allows clients to receive acknowledgement that a given write was committed. WriteAck() checks for a committed replay log entry, returns the result to the client, and if found in the replay log it garbage collects the ACKed entry and older entries from that client. It also propagates the removals from the replay log upstream using AckReplayLog(). For the pending list, sent list, and replay log utilized by message propagation functions, we use a combination of priority queue ordering and maps in order to ensure that request sequence numbers are processed in the correct order and allowing us to easily seek out specific entries for garbage collection, acknowledgements, and avoiding sequence gaps.

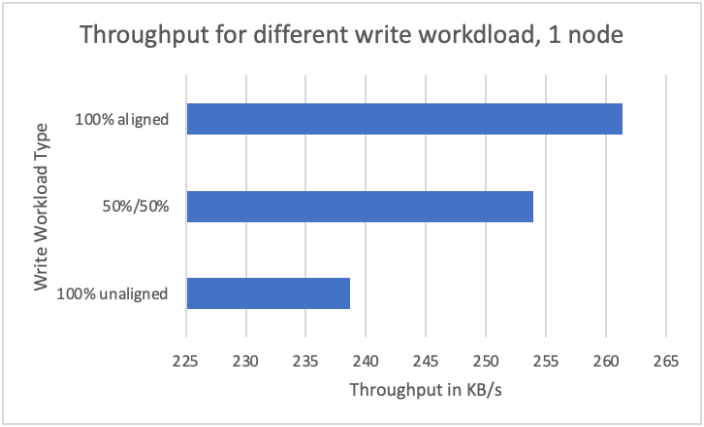


The diagrams above visualize the flow of message propagation for a typical write, write acknowledgement, and read request from a client.

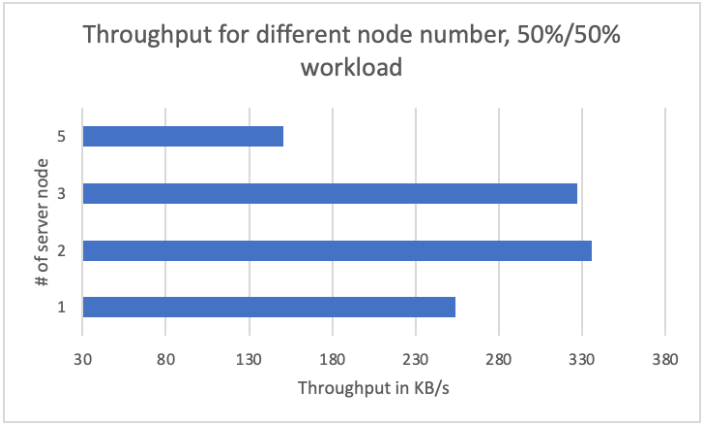
The measurements of round-trip write latency shown here are in line with our expectations, with latency increasing at an approximately linear interval proportional to the number of nodes added. Each additional node added to the chain adds the time overhead of an additional relayWrite() and



relayWriteAck() call when doing upstream and downstream propagation, shown in the earlier message propagation diagram. Time overhead incurred by special head and tail logic such as the tail node's 'commit' background thread is included in a single-server system with 1 node, and thus we assume all additional nodes will only add the independent write/acknowledgement propagation time present on a 'mid' state node (common to all node states). The fact that we observe linear write latency increases in our benchmark demonstrates that this assumption is true, and that our system does not suffer from other overheads which might compound on each other as we scale up. All latency measurements were taken from the minimum value of 10000 runs on the CloudLab testbed.



Comparison of throughput between different workload



Comparison of throughput between different number of chain length

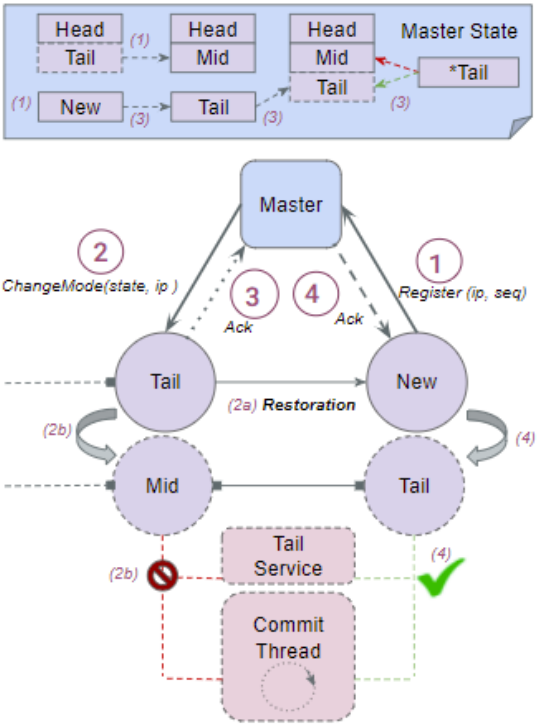
Throughput experiment set-up:
The workload in each run of the throughput experiment consists of 4000 requests sent from 40 clients. Each client is an individual process that sends a specified number of aligned/unaligned writes(100) sequentially, and are started as background jobs using bash script to ensure maximized concurrency. Several runs of the same experiment condition are performed and the highest observed throughput value was taken. Throughput is measured using timestamps recorded on the tail node. Total time elapsed is taken using the first and last recorded timestamp of the tail observed.

Result Interpretation:
The throughput graph for different workload conditions when with 1 node server fits our expectation that workload with more aligned writes will have better throughput, since it takes faster for the storage to process aligned writes compared to unaligned writes.

The throughput graph for different node numbers does not fit our expectation exactly. We predicted that the throughput for servers with more than one node would be higher, since the job of head and tail are separated into two machines, however we did not predict the throughput

would drop significantly for servers with 5 nodes. Due to time constraints we did not investigate why it was the case. If we do more investigations, one thing to do is to check if there is any bottleneck between node and node communication, and whether it limits the concurrent requests that reach the tail node.

Integration



The figure on the right outlines the integration process necessary to bring new nodes into the chain. The first step involves the node sending a register() call to the master containing their IP address and last committed sequence number. The master generates a virtual node, and updates the state of the current tail internally. Next, the master uses a changeMode() call to inform the current tail of its new state, the new node's IP address, and its last seen sequence number. The tail pauses message propagation and builds communication with the new node then initiates the restore process.

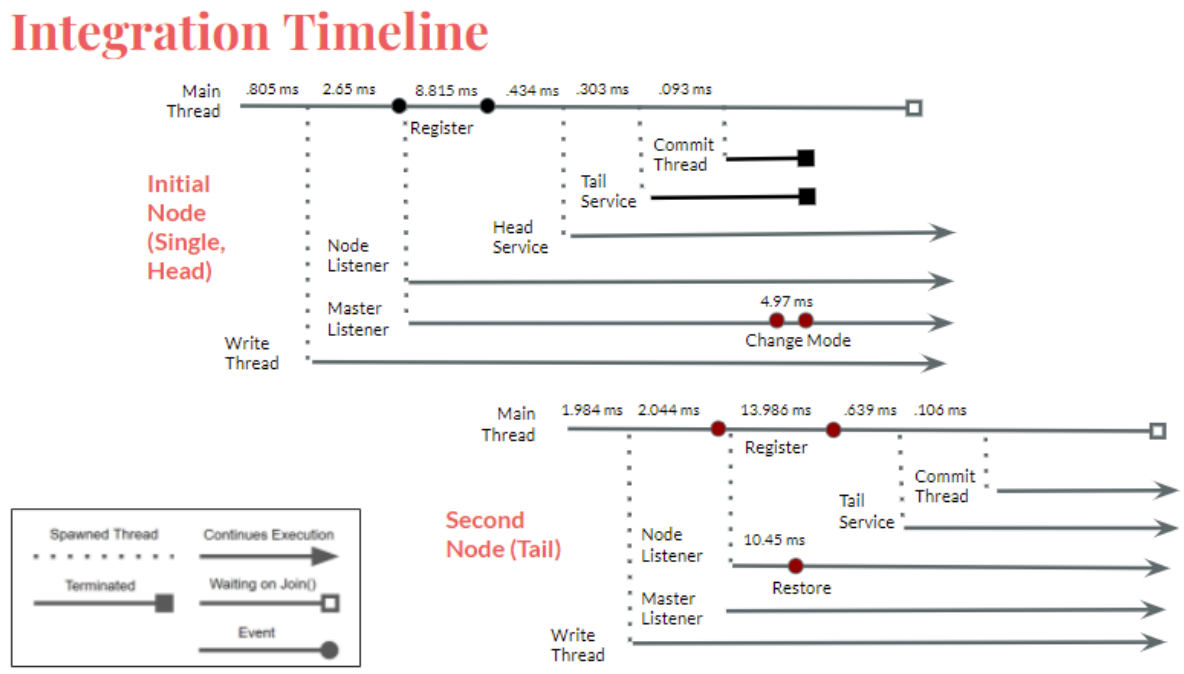
During the restoration phase of integration the new node must be brought up to speed with the state of the rest of the chain. New nodes are permitted to come up with existing writes if they were previously part of the chain, or they can be brought up clean. The current tail uses a series of gRPC calls to send missing writes, a copy of the replay log, and its active sent list.

After the new node has been restored to the appropriate state the current tail kills the commit thread and tail service, updates its own state, and returns to the master. The master finishes registration by updating the joining node's virtual state to tail, adding it to the nodelist, and updating the tail pointer. It then returns the call with the IP address of the old tail. Finally, the new node builds communication with the old tail, updates its own state to tail, and launches the commit thread and tail services.

If the chain is empty when the new node is registertered then step 2 in the diagram is skipped. The master sets the node's virtual state to ‘single server’, sets the head pointer in addition to the tail pointer, and returns an empty IP address to the new node. On seeing this the new node sets its own state to ‘single server’ and launches the head service in addition to the tail services.

In addition to registration, nodes launch a series of services to facilitate communication regardless of state. These include a ‘write thread’ that is responsible for propagating messages down the chain, and listening services allowing for inter-node communication and messages from the master. But the figure below shows that the majority of the integration timeline is taken up by the registration process. This timeline shows registration when both nodes are brought up

clean. However, the time necessary for the restoration process increments linearly as the number of writes that must be sent grows.

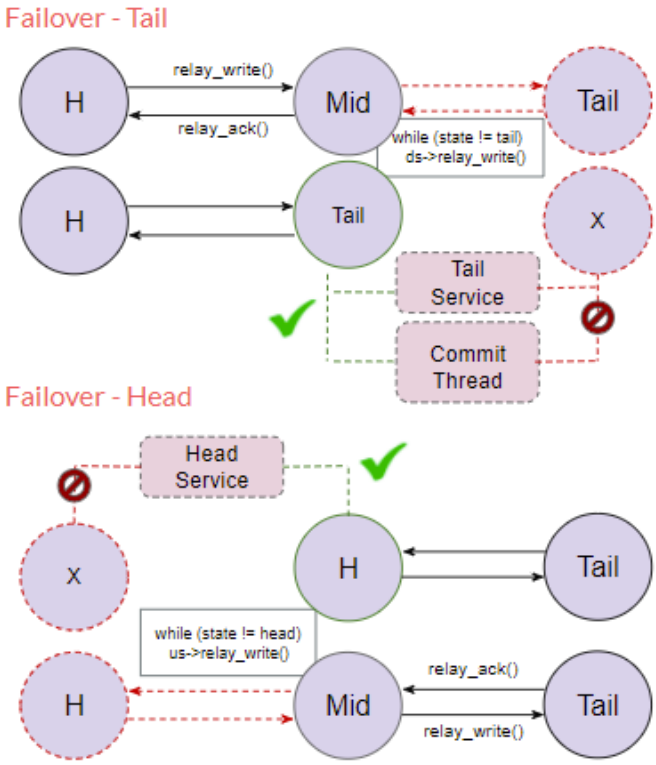


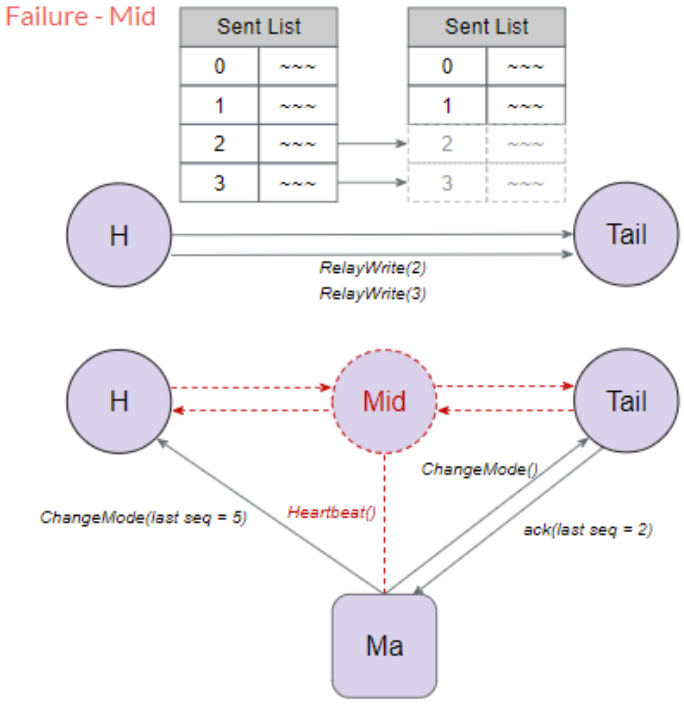
Failure

The master runs a heartbeat thread to identify node failures. When the thread wakes up it iterates through the virtual node list and sends heartbeats to each server. If the master detects failure it will identify replacement nodes, remove bad nodes from its node list, and update virtual node state. During this step the master will move down the chain towards endpoints verifying that the next node is available before updating state. This allows the master to detect multiple concurrent failures in the chain in a single correction. Once accessible nodes are identified, the impacted nodes are sent changeMode() calls and update their state accordingly. When failure detection is complete the heartbeat is run again immediately to detect disjoint failures. Disjoint failures are handled in separate correction steps and could result in multiple state changes for an impacted node.

Endpoint failures represent the simplest case for recovery. Both head and tail failure follow similar mechanisms, and have minimal impact. A diagram of these mechanisms is found below. The node that takes over the endpoint may have had message propagation in progress via the write and commit threads. These relay calls will continue to contact the previous endpoint until their state is updated. The calls are wrapped in state checks, and as soon as the state is updated the check will break out of the communication cycle. On head failure nothing else needs to happen. On tail failure, the new tail will launch the commit thread as part of the state change and will start committing the sent list. Figure A visualizes this process.

Mid-node failure requires inter-node communication to resolve. The master notifies the downstream node of the failure, and sends the new upstream nodes ip address. This node builds communication upstream and returns the highest sequence number it has in its sent list to the master. The master sends this information to the upstream node as part of the changeMode() call. This node identifies missing writes based on its own sent list and restarts the message propagation process with relayWrite() calls.





Consistency

To evaluate consistency and generate demos, we utilized the crash point methodology and checksum functions mentioned in our experimental setup for simple debugging and more extended client demo tests.

We have included four demo videos in our presentation, which are also included in the linked folder:

https://drive.google.com/drive/folders/1y1STB9Z9zsc5_6quiDZosHFA7gRL3UF3?usp=sharing.

We demonstrated successful and consistent failover in the cases of head failure, mid failure, and tail failure; in each case, a client performed and ACKed a normal write on the 3-node system, deliberately crashed the given target node with a write that would be partially written on preceding nodes and need to be fully written and committed after failover, and finally performed a write on the operational 2-node system. We then compared checksums to show that both remaining servers had successfully committed the data and remained consistent across all writes. In all demos, we crash the server mid-write in the ‘write’ background thread.

In our fourth demo, we crash the head node and perform writes similar to our setup for the head failure case; however, we then bring the old head back up and have it re-integrate as a new tail, requiring it to catch up on missed writes via reintegration communication with the old tail. We also perform an additional write across the recovered 3-node system, and then show that all rounds of writes (including from the mid-write crash and from when the node was offline) are present and consistent across the chain.

Conclusion

Our goal was to create a replicated block store allowing for an arbitrary number of replication nodes while providing strong consistency. We optimized our system for write throughput, and reads/writes aligned to 4K block boundaries. We designed our system to be modular, so that we could develop multiple parts independently. We did not leave quite enough time at the end. While we were able to debug putting the pieces together and perform some testing and benchmarking, we would have preferred to do more. The tests and benchmarks we did perform do show our system provides strong consistency and the performance we expected.